Ficha 5

Programação Imperativa

1 Ficheiros de acesso sequencial

1. Considere o seguinte tipo para representar uma lista ligada de inteiros:

```
typedef struct lista {
    int valor;
    struct lista *prox;
} *LInt;
```

   Defina funções `int writeLInt (FILE *f, LInt l)` e `int readLInt (FILE *f, LInt *l)` de escrita e leitura de uma destas listas em ficheiro de forma a garantir que a leitura de uma lista previamente escrita, preserva a ordem dos elementos na lista.

   As funções devem retornar o número de items escritos/lidos.

2. Suponha agora que pretendemos armazenar listas de *strings* (com tamanho variável).

```
typedef struct slstr {
    char  *valor;
    struct slstr *prox;
} NLstr, *Lstr;
```

   (a) Defina uma função `char * readStr (FILE *f, char t)` que lê do ficheiro `f` uma string terminada pelo caracter `t` (ou pelo fim do ficheiro). A função deverá alocar o espaço necessário para armazenar a string lida.

   (b) Defina funções `int writeLstr (FILE *f, Lstr l)` e `int readLstr (FILE *f, Lstr *l)` de escrita e leitura de uma lista de strings em ficheiro de forma a garantir que a leitura de uma lista previamente escrita, preserva a ordem dos elementos na lista.

3. Considere a seguinte definição de uma árvore binária em que cada nodo contem um inteiro e uma *string* de tamanho variável.

```
typedef struct spares {
   int n; char *s;
   struct spares *esq, *dir;
} NPar, *ABPares;
```

   Defina funções de escrita e leitura de uma destas árvores em ficheiro. Note que, para que estas operações preservem a forma da árvore é necessário que os nodos da árvore sejam escritos seguindo uma travessia *preorder*.

2 Estruturas ligadas em ficheiro

Nas alíneas anteriores, o problema da persistência dos dados de um programa foi resolvida providenciando funções que armazenam os dados num ficheiro e que os lêem de ficheiro. Esta solução é viável (e provavelmente indicada) quando se trata de estruturas de pequena dimensão que podem ser totalmente armazenadas em memória e cujo tempo de leitura ou escrita não a torne inviável.
Uma solução alternativa consiste em armazenar os dados apenas em ficheiro (de leitura/escrita). Aqui torna-se indispensável que continuemos a ter os dados organizados nessas mesmas estruturas ligadas de forma a melhorar o comportamento das várias operações.
Considere o seguinte tipo de dados para representar um contacto:

```
typedef struct contacto {
   char nome[80];
   char email[80];
   char telefone[10];
} Contacto;
```

1. Para armazenar os contactos num ficheiro, organizados numa lista ordenada pelo nome, vamos estabelecer o seguinte:

   - No início do ficheiro vamos armazenar o endereço onde está armazenado o primeiro contacto.
   - Para cada contacto, armazenamos também o endereço onde está armazenado o contacto seguinte. Se se tratar do último contacto, esse valor será `0L`. Para isso vamos definir o seguinte tipo:

     ```
     typedef struct registo {
         Contaco c;
         long proximo;
     } Registo;
     ```

   - Um novo registo será sempre escrito no final do ficheiro (e ligado convenientemente na lista).

   Defina as seguintes funções:

   (a) `FILE *abreFich (char *nome)` que abre o ficheiro `fich`. Caso ele não exista, deverá ser criado de forma a obedecer aos pressupostos apresentados.

   (b) `long novo (FILE *f)` que devolve o endereço onde deve ser escrito um novo registo no ficheiro.

   (c) `long primeiro (FILE *f)` que devolve o endereço onde está armazenado o primeiro contacto (`0L` se não houver nenhum).

   (d) `void dumpLista (FILE *f)` que escreve no ecran a lista (ordenada) dos contactos.

   (e) `int lookup (Contacto *dest, FILE *f, char *nome)` que procura a informação relatva ao nome `nome`. A função deverá retornar `0` em caso de sucesso (preenchendo `dest`).

(f) `int acrescenta (FILE *f, char *nome, char *email, char *telef)` que acrescenta um novo contacto. Se o nome já existir, a função deve actualizar os dados (email e telfone). Nesse caso deve retornar `1`, caso contrário deverá retornar `0`.

2. A solução descrita acima pressupõe que não se fazem remoções. Se tal não for o caso devemos organizar as posições apagadas numa lista de forma a serem reaproveitadas em futuras inserções. Para isso vamos armazenar no início do ficheiro a informação sobre o endereço da primeira posição apagada, bem como o do primeiro contacto. As células apagads deverão estar elas próprias ligadas (uma *stack* de posições apagadas)

```
typedef struct controlo {
    long primeiro;
    long apagados;
} Controlo;
```

   (a) Redefina a função `FILE *abreFich (char *nome)` que abre o ficheiro `fich`. Caso ele não exista, deverá ser criado de forma a obedecer aos pressupostos apresentados.

   (b) Redefina a função `long novo (FILE *f)` que devolve o endereço onde deve ser escrito um novo registo no ficheiro. Se o endereço devolvido for de um dos registos previamente apagados, esse endereço deve passar a ser considerado usado!

   (c) Defina a função `void liberta (FILE *f, long end)` que marca como livre o endereço `end` do ficheiro `f`.

   (d) Defina a função `int remove (FILE *f, char *nome)` que remove da lista de contactos a informação sobre `nome`. A função devolve um código de erro (`0` em caso de sucesso).

A Funções de manipulação de ficheiros

As operações de acesso a ficheiros (*buffered IO*) que vamos usar são:

- `FILE *fopen(char filename[], char mode[]);` que abre o ficheiro de nome `filename`. O modo de acesso é especificado pela *string* `modo` e pode ser:
  - `"w"` o ficheiro é aberto para escrita. Se não existir é criado e se existir o seu conteúdo é apagado.
  - `"r"` o ficheiro é aberto para leitura na primeira posição. Se o ficheiro não existir a função retorna `NULL` e não é criado nenhum ficheiro.
  - `"a"` o ficheiro é aberto para escrita no fim. Cria o ficheiro se ele não existir e preserva o seu conteúdo se ele existir. Todas as operações de escrita são feitas no fim do ficheiro.
  - `"w+"` o ficheiro é aberto para escrita e leitura. Se não existir é criado e se existir o seu conteúdo é apagado.
  - `"r+"` o ficheiro é aberto para leitura e escrita na primeira posição. Se o ficheiro não existir a função retorna `NULL` e não é criado nenhum ficheiro.

  - `"a+"` o ficheiro é aberto para escrita no fim e leitura. Cria o ficheiro se ele não existir e preserva o seu conteúdo se ele existir. Todas as operações de escrita são feitas no fim do ficheiro.

  Em caso de erro a função retorna `NULL`

- `int fclose(FILE *f);` que fecha o ficheiro, actualizando o seu conteúdo. Retorna `0` em caso de sucesso.
- `int fprintf(FILE *f, char format[], ...);` escreve na posição actual do ficheiro a *string* resultante da conversão para texto dos vários argumentos segundo o formato especificado. Retorna o número de items escritos.
- `int fscanf(FILE *f,  char format[], ...);` lê, a partir da posição actual do ficheiro de texto, os items segundo o formato especificado. retorna o número de items lidos.
- `int fgetc(FILE *f);` lê da posição actual do ficheiro um caracter (*byte*). Retorna o valor `EOF` se essa posição corresponder ao fim do ficheiro.
- `int fputc(int c, FILE *f);` escreve na posição actual um caracter (*byte*).
- `size_t fread(void *ptr, size_t size, size_t nitems, FILE *f);` transfere para a posição de memória `ptr` os prósimos `size * nitems` bytes do ficheiro `f`.
- `size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *f);` escreve a partir da posição actual os `size * nitems` bytes que se encontram a partir do endereço `ptr`. Retorna o número de items que foram escritos
- `int fseek(FILE *f, long offset, int base);` passa a posição actual do ficheiro para `offset` a partir do referencial `base`. Este último pode ter os seguintes valores:
  - `SEEK_SET` significando `offset`s relativos ao início do ficheiro.
  - `SEEK_END` significando `offset`s relativos ao fim do ficheiro.
  - `SEEK_CUR` significando `offset`s relativos à posição actual.

  Retorna `0` em caso de sucesso.
- `long ftell(FILE *f);` retorna a posição actual do ficheiro.